Protecting Encrypted Cookies from Compression Side-Channel Attacks

Janaka Alawatugoda[1] Douglas Stebila[1,2] Colin Boyd[3]

[1] *School of Electrical Engineering and Computer Science,*
[2] *Mathematical Sciences School,*
Queensland University of Technology, Brisbane, Australia
janaka.alawatugoda@qut.edu.au, stebila@qut.edu.au
[3] *Department of Telematics, Norwegian University of Science and Technology,*
Trondheim, Norway
colin.boyd@item.ntnu.no

Abstract

Compression is desirable for network applications as it saves bandwidth; however, when data is compressed before being encrypted, the amount of compression leaks information about the amount of redundancy in the plaintext. This side channel has led to successful real-world attacks (the CRIME and BREACH attacks) on web traffic protected by the Transport Layer Security (TLS) protocol. The general guidance in light of these attacks has been to disable compression, preserving confidentiality but sacrificing bandwidth. In this paper, we examine two techniques—heuristic separation of secrets and fixed-dictionary compression—for enabling compression while protecting high-value secrets, such as cookies, from attack. We model the security offered by these techniques and report on the amount of compressibility that they can achieve.

Contents

1 Introduction

To save communication costs, network applications often compress data before transmitting it; for example, the Hypertext Transport Protocol (HTTP) [9, §4.2] has an optional mechanism in which a server compresses the body of an HTTP response, most commonly using the gzip algorithm. When encryption is used to protect communication, compression must be applied before encryption (since ciphertexts should look random, and thus have little apparent redundancy that can be compressed). In fact, to facilitate this, the Transport Layer Security (TLS) protocol [7, §6.2.2] has an optional compression mode that will compress all application before encrypting it.

While compression is useful for reducing the size of transmitted data, it has had a negative impact when combined with encryption, because the amount of compression acts as a *side channel*. Most research considers side-channels that leak information like timing [15, 6, 5] or power consumption [12, 16], which can reveal information about cryptographic operations and secret parameters.

1.1 Compression-based leakage.

In 2002, Kelsey [14] showed how compression can act as a form of side-channel leakage. If plaintext data is compressed before being encrypted, the length of the ciphertext reveals information about the amount of compression, which in turn can reveal information about the plaintext. Kelsey notes that this side channel differs from other types of side channels in two key ways: "it reveals information about the plaintext, rather than key material", and "it is a property of the algorithm, not the implementation".

Kelsey's most powerful attack is an *adaptive chosen input attack*: if an attacker is allowed to choose inputs x that are combined with a target secret s and $x\|s$ is compressed and encrypted, then observing the length of the outputs can eventually allow the attacker to extract the secret s. For example, the to determine the first character of s, the attacker could ask to have the string $x = \texttt{prefix*prefix}$ combined with s compressed and encrypted, for every possible character `*`; in one case, when $\texttt{*} = s_1$, the amount of redunancy is higher and the ciphertext should be shorter. Once each character of s is found, the attack can be carried out on the next character. The attack is somewhat noisy, but succeeds reasonably often.

Key to this attack is the fact that most compression algorithms (such as the DEFLATE algorithm underlying gzip) are adaptive: they adaptively build and maintain a *dictionary* of recently observed strings, and replace subsequent occurences of that string with a code.

1.2 The CRIME and BREACH attacks.

In 2012, Rizzo and Duong [19] showed how to apply Kelsey's adaptive chosen input attack against gzip compression as used in TLS, in what they called the Compression Ratio Info-leak Mass Exploitation (CRIME) attack. The primary target of the CRIME attack was the user's cookie in the HTTP header. If the victim visited an attacker-controlled web page, the attacker could use Javascript to cause the victim to send HTTP requests to URLs of the attacker's choice on a specified server. The attacker could adaptively choose those URLs to include a prefix to carry out Kelsey's adaptive chosen input attack. Some care is required to ensure the padding does not hide the length with block ciphers, but this can be dealt with.

As a result of the CRIME attack, it was recommended that TLS compression be disabled, and the Trustworthy Internet Movement's SSL Pulse report for September 2014 indicates less than 10% of websites have TLS compression enabled [22]; moreover, all major browsers have disabled it.

However, compression is also built into the HTTP protocol: servers can optionally compress the body of HTTP responses. While this excludes the cookie in the header, this type of attack can still be successful against secret values in the HTTP body, such as anti-cross-site request forgery (CSRF) tokens. Suggested by Rizzo and Duong, this was demonstrated Gluck et al. [10] in the so-called Browser Reconnaissance and Exfiltration via Adaptive Comrpession of Hypertext (BREACH) attack.

1.3 Mitigation techniques.

Gluck et al. [10] discussed several possible mitigation techniques against the BREACH attack, listed in decreasing order of effectiveness:

1. Disabling HTTP compression
2. Separating secrets from user input
3. Randomizing secrets per request

4. Masking secrets (effectively randomizing by XORing with a random secret perrequest)
5. Protecting vulnerable pages with CSRF
6. Length hiding (by adding random number of bytes to the responses)
7. Rate-limiting the requests

Despite the demonstrated practicality of the BREACH attack, support for and use of HTTP compression remains widespread, due in large part to the utility of decreasing communication costs and time. In fact, compression is even more tightly integrated into the proposed HTTP version 2 [4] than previous versions. Techniques 2–4 generally require changes to both browsers and web servers. For example, masking secrets such as anti-CSRF tokens requires new mark-up for secrets, which browsers and servers can interpret and apply the randomized masking technique. Techniques 5–7 can be unilaterally applied by web servers, though length hiding can be defeated with statistical averaging, and rate-limiting must find a balance between legitimate requests and the amount of information obtainable.

1.4 Related work.

There has been little academic study of compression and encryption. Besides Kelsey's adaptive chosen input attack and the related CRIME and BREACH attacks, the only relevant work we are aware of is that of Kelley and Tamassia [13]. They give a new security notion called *entropy-restricted semantic security* (ER-IND-CPA) for *keyed compression functions* which combine both encryption and compression: compared with the normal indistinguishability under chosen plaintext attack (IND-CPA) security notion, in ER-IND-CPA the adversary should be able to distinguish between the encryption of two messages that *compress* to the same length. Kelley and Tamassia then show how to construct a cipher based on the LZW compression algorithm by rerandomizing the compression dictionary. Unfortunately, the ER-IND-CPA notion does not capture the CRIME and BREACH attacks, which depend on observing messages that compress to different lengths.

In leakage-resilient security definitions [1, 2, 8, 17, 11], leakage of the secret key is addressed. This differs from the setting in compression-based side-channel attacks, which addresses leakage of the plaintext. Thus, previous leakage-resilient security definitions are not suitable to model compression-based side-channel attacks.

1.5 Our contributions.

In this work, we study symmetric-key compression-encryption schemes, with the characterizing the security properties that can be achieved by various mitigation techniques in the face of CRIME-/BREACH-like attacks.

To some extent, the side channel exposed by compression is fundamentally unavoidable: if transmission of data is decreased, nothing can hide the fact that some redundancy existed in the plaintext. Hence, we focus our study on the ability of the attacker to learn specific "high value" secrets embedded in a plaintext, such as cookies or anti-CSRF tokens. In our models, we imagine there is a secret value ck, and the adversary can adaptively obtain encryptions

$$\mathrm{E}_k(m' \| ck \| m'') \tag{1}$$

for messages m' and m'' of his choosing.

The first mitigation technique we consider is that of *separating secrets*. During compression/encryption, an application-aware filter is applied to the plaintext to separate out any potential secret values from the data, the remaining plaintext is compressed, then the secrets and compressed plaintext are encrypted; after decryption, the inverse of the filter is used to reinsert the secret values in the plaintext. Assuming the filter fully separates out all secret values, we show that the separating secrets technique is able to achieve a strong notion of protection, which we call *chosen cookie indistinguishability* (CCI): the adversary cannot determine which of two cookies ck_0 and ck_1 of the adversary's choice was encrypted with messages of the adversary's choice given ciphertexts as in equation (1).

The second mitigation technique we consider is the use of a *fixed-dictionary compression scheme*, where the dictionary used for compression does not adapt to the plaintext being compressed, but instead is preselected in advance based on the expected distribution of plaintext messages, for example including

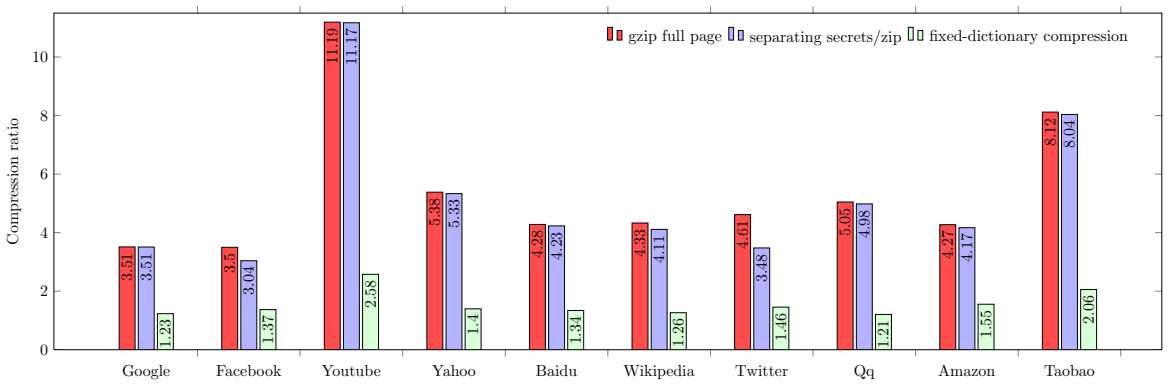

Figure 1: Compression ratios of Gzip compression and mitigation techniques

common English words like "the" and "and".[1] We show that, if the secret values are sufficiently high entropy, then fixed-dictionary compression is able to achieve *cookie recovery* (CR) security: if the secret cookie chosen uniformly at random, the adversary cannot recover the entire secret cookie even given an adaptive message attack as in equation (1). While cookie recovery security does not meet the "gold standard" of indistinguishability notions for encryption, it may be sufficient for some settings, and would for example protect compressed HTTP traffic from CRIME and BREACH attacks attempting to recover cookies and anti-CSRF tokens.

We also characterize the relationship among the CCI and CR security notions, as well as an intermediate notion called *random cookie indistinguishability* (RCI) and the ER-IND-CPA notion of Kelley and Tamassia [13].

In the separating secrets technique, if the number of secrets extracted by the separating filter is relatively small, then the compressibility generally remains close to that of normal compression of the full plaintext. In the fixed-dictionary compression technique, compressibility suffers quite a bit compared to adaptive techniques on the full plaintext, although if the dictionary is constructed from a corpus of text similar to the plaintext, then some compression can be achieved. In Figure 1, we report experimental results comparing compression ratios for these two techniques on the HTML, CSS, and Javascript source code of the top 10 global websites as reported by Alexa Top Sites[2]. On average, the compression ratio (uncompressed size : compressed size) of the gzip algorithm applied to the full source code was 5.42×; applying a separation filter that extracted all values following `value=` in the HTML source code yielded an average compression ratio of 5.20×; compression of each page using a fixed dictionary trained on all 10 pages yielded an average compression ratio of 1.55×.

2 Definitions

Notation. If x is a string, then $x_{i:\ell}$ denotes the length-ℓ substring of x starting at position i: $x_{i:\ell} = x_i \| \ldots \| x_{i+\ell-1}$. If x and y are strings, then $x \subseteq y$ denotes that x is a substring of y. The *index* of x in y is the smallest i such that $y_{i:|x|} = x$.

2.1 Encryption and compression schemes

Recall the standard definition of an encryption scheme:

Definition 2.1 (Symmetric-key encryption scheme)**.** A *symmetric-key encryption scheme* Π for message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$ is a tuple of algorithms:

- KeyGen() $\xrightarrow{\$} k$: A probabilistic *key generation algorithm* that generates a random key k in the keyspace $\mathcal{K}$.
- Enc$(k, m) \xrightarrow{\$} c$: A possibly probabilistic *encryption algorithm* that takes as input a key $k \in \mathcal{K}$ and a message $m \in \mathcal{M}$ and outputs a ciphertext $c \in \mathcal{C}$.

[1] Sadly "cryptography" is only the 29,697th most-frequently used English word. (http://en.wiktionary.org/wiki/Wiktionary:Frequency_lists/PG/2006/04/20001-30000)

[2] http://www.alexa.com/topsites

$\underline{\text{Exp}^{\text{IND-CPA}}_{\Pi}(\mathcal{A})}$

1: $k \xleftarrow{\$} \text{KeyGen}()$
2: $b \xleftarrow{\$} \{0,1\}$
3: $(m_0, m_1) \xleftarrow{\$} \mathcal{A}^{\text{Enc}_k(\cdot)}()$
4: **if** $|m_0| \neq |m_1|$, **then return** 0
5: $c \leftarrow \text{Enc}_k(m_b)$
6: $b' \xleftarrow{\$} \mathcal{A}^{\text{Enc}_k(\cdot)}(c)$
7: **if** $b' = b$ **then**
8: **return** 1
9: **else**
10: **return** 0

$\underline{\text{Exp}^{\text{ER-IND-CPA}}_{\Pi\circ\Gamma,\mathcal{L}}(\mathcal{A})}$

1: $k \xleftarrow{\$} \text{KG}()$
2: $b \xleftarrow{\$} \{0,1\}$
3: $(m_0, m_1) \xleftarrow{\$} \mathcal{A}^{\text{E}_k(\cdot)}()$
4: **if** $m_0 \notin \mathcal{L}$ or $m_1 \notin \mathcal{L}$, **then return** 0
5: $c \leftarrow \text{E}_k(m_b)$
6: $b' \xleftarrow{\$} \mathcal{A}^{\text{E}_k(\cdot)}(c)$
7: **if** $b' = b$ **then**
8: **return** 1
9: **else**
10: **return** 0

Figure 2: Security experiments for indistinguishability under chosen plaintext attack (IND-CPA, left) and entropy-restricted IND-CPA (ER-IND-CPA, right)

- $\text{Dec}(k, c) \to m'$ or $\perp$: A deterministic *decryption algorithm* that takes as input a key $k \in \mathcal{K}$ and a ciphertext $c \in \mathcal{C}$, and outputs either a message $m' \in \mathcal{M}$ or an error symbol $\perp$.

Correctness of symmetric-key encryption is defined in the obvious way: for all $k \xleftarrow{\$} \text{KeyGen}()$, for all $m \in \mathcal{M}$, we require that $\text{Dec}(k, \text{Enc}(k, m)) = m$.

Definition 2.2 (Compression scheme)**.** A *compression scheme* Γ for message space $\mathcal{M}$ with output space $\mathcal{O}$ is a pair of algorithms:

- $\text{Comp}(m) \to o$: A *compression* algorithm that takes as input a message $m \in \mathcal{M}$ and outputs an encoded value $o \in \mathcal{O}$.
- $\text{Decomp}(o) \to m'$ or $\perp$: A *decompression* algorithm that takes as input an encoded value $o \in \mathcal{O}$ and outputs a message $m' \in \mathcal{M}$ or an error symbol $\perp$.

Note that $|\text{Comp}(m)|$ may not necessarily be less than $|m|$; Shannon's coding theorem implies that no algorithm can encode every message with shorter length, so not all messages may actually be "compressed", and in fact some may be length-increased.

Correctness of a compression scheme is again defined in the obvious way: for all $m \in \mathcal{M}$, we require that $\text{Decomp}(\text{Comp}(m)) = m$.

In this paper, we are interested in *symmetric-key compression-encryption schemes* that come from the composition of a compression scheme and an encryption scheme.

Definition 2.3 (Symmetric-key compression-encryption scheme)**.** Let $\Gamma = (\text{Comp}, \text{Decomp})$ be a compression scheme with message space $\mathcal{M}$ and output space $\mathcal{O}$. Let $\Pi = (\text{KeyGen}, \text{Enc}, \text{Dec})$ be a symmetric-key encryption scheme with message space $\mathcal{O}$ and ciphertext space $\mathcal{C}$. The *symmetric-key compression-encryption scheme* $\Pi \circ \Gamma$ constructed from Γ and Π is a tuple of the following algorithms:

$$\begin{aligned} \text{KG}() &= \Pi.\text{KeyGen}() \\ \text{E}_k(m) &= \Pi.\text{Enc}_k(\Gamma.\text{Comp}(m)) \\ \text{D}_k(c) &= \Gamma.\text{Decomp}(\Pi.\text{Enc}_k(c)) \end{aligned}$$

Note that $\Pi \circ \Gamma$ is itself a symmetric-key encryption scheme with message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$. Moreover, if Γ and Π are both correct, then so is $\Pi \circ \Gamma$.

2.2 Existing security notions

The standard security notion for symmetric-key encryption is indistinguishability of encrypted messages. In this paper, we focus on chosen plaintext attack. The security experiment $\text{Exp}^{\text{IND-CPA}}_{\Pi}(\mathcal{A})$ for indistinguishability under chosen plaintext attack (IND-CPA) of a symmetric-key encryption scheme Π against a stateful adversary $\mathcal{A}$ is given in Figure 2. The *advantage* of $\mathcal{A}$ in breaking the IND-CPA experiment for Π is $\text{Adv}^{\text{IND-CPA}}_{\Pi}(\mathcal{A}) = \left|\Pr\left(\text{Exp}^{\text{IND-CPA}}_{\Pi}(\mathcal{A}) = 1\right) - 1/2\right|$.

Kelley and Tamassia [13] give a definition of *entropy-restricted* IND-CPA *security* which applies to compession-encryption schemes $\Pi \circ \Gamma$, and demands indistinguishability of messages from the same class $\mathcal{L} \subseteq \mathcal{M}$; typically, $\mathcal{L}$ is the class of messages that compress to the same length under Γ.Comp, such as:

$$\mathcal{L}_\ell = \{m \in \mathcal{M} : |\text{Comp}(m)| = \ell\} \quad .$$

The ER-IND-CPA security experiment is given in Figure 2. (We adapt their notion slightly to use our composition notation in Definition 2.3.) Kelley and Tamassia note that any IND-CPA-secure symmetric-key encryption scheme Π, combined with any compression scheme Γ, is immediately ER-IND-CPA-secure.

2.3 New security notions

We focus on the ability of an attacker to learn about a secret piece of data inside a larger piece of data, where the attacker controls everything except the secret data. We use the term *cookie* to refer to the secret data; in practice, this could be an HTTP cookie in a header, an anti-cross-site request forgery (CSRF) token, or some piece of personal information. We will allow the attacker to adaptively obtain encryptions of compressions of data of the form $m' \| ck \| m''$ for a secret cookie ck and adversary-chosen message prefix m' and suffix m''.

We now present three notions for the security of cookies in the context of compression-encryption schemes:

- *Cookie recovery* (CR) *security*: A simple, but relatively weak, security notion for symmetric-key compression-encryption schemes is that of *cookie recovery*: it should be hard for the attacker to recover a secret value, even given adaptive access to an oracle that encrypts plaintexts of his choosing with the target cookie embedded.
- *Random cookie indistinguishability* (RCI) *security*: Here, the adversary has to decide which of two randomly chosen cookies was embedded in the encrypted plaintext, even given adaptive access to an oracle that encrypts plaintexts of his choosing with the target cookie embedded. RCI includes indistinguishability of messages.
- *Chosen cookie indistinguishability* (CCI) *security*: Here, the adversary has to decide which of two cookies of the adversary's choice was embedded in the encrypted plaintext, even given adaptive access to an oracle that encrypts plaintexts of his choosing with the target cookie embedded. CCI includes indistinguishability of messages.

These security notions are formalized in the following definition, which refers to the security experiments shown in Figure 3.

Definition 2.4 (CR, RCI, CCI security)**.** Let $\Pi \circ \Gamma$ be a compression-encryption scheme constructed from symmetric-key encryption scheme Π and compression scheme Γ. Let $\mathcal{A}$ denote an algorithm. Let $\mathcal{CK}$ denote the cookie space. Let $\mathsf{xxx} \in \{\mathsf{CR}, \mathsf{RCI}, \mathsf{CCI}\}$ be a security notion. Consider the security experiment $\text{Exp}^{\mathsf{xxx}}_{\Pi\circ\Gamma,\mathcal{CK}}(\mathcal{A})$ defined in Figure 3. Let $\text{Succ}^{\mathsf{CR}}_{\Pi\circ\Gamma,\mathcal{CK}}(\mathcal{A}) = \Pr\left(\text{Exp}^{\mathsf{CR}}_{\Pi\circ\Gamma,\mathcal{CK}}(\mathcal{A}) = 1\right)$ denote the probability that $\mathcal{A}$ wins the cookie recovery experiment for $\Pi \circ \Gamma$ and $\mathcal{CK}$. Similarly, let $\text{Adv}^{\mathsf{xxx}}_{\Pi\circ\Gamma,\mathcal{CK}}(\mathcal{A}) = \left|\Pr\left(\text{Exp}^{\mathsf{xxx}}_{\Pi\circ\Gamma,\mathcal{CK}}(\mathcal{A}) = 1\right) - 1/2\right|$, $\mathsf{xxx} \in \{\mathsf{RCI}, \mathsf{CCI}\}$ denote the advantage that $\mathcal{A}$ has in winning the random cookie and chosen cookie indistinguishability experiments.

The motivation for incorporating the message indistinguishability property is as follows: in real-world situations, all the data in the application layer (messages and secrets) are sent to the security layer (SSL/TLS) for encryption. In the security layer, encryption happens on all the data coming from the application layer. If we only consider the indistinguishability of the cookie, a IND-CPA-secure encryption scheme which only encrypts the cookie can satisfy the cookie indistinguishability requirement, which does not address the real-world situation because the security layer does not encrypt just the cookie, instead encrypts all the data (message and cookie) coming from the application layer. Therefore, in security notions we need to make sure that the encryption scheme encrypts all the data. Thus, notions combining both message indishtinguishability and cookie indistinguishability provide stronger security.

$\underline{\mathrm{Exp}^{\mathsf{CR}}_{\Pi\circ\Gamma,\mathcal{CK}}(\mathcal{A})}$

$k \xleftarrow{\$} \mathrm{KG}()$
$ck \xleftarrow{\$} \mathcal{CK}$
$ck' \xleftarrow{\$} \mathcal{A}^{\mathrm{E}_k(\cdot\|ck\|\cdot),\mathrm{E}_k(\cdot)}()$
if $ck' = ck$ **then**
  return 1
else
  return 0

$\underline{\mathrm{Exp}^{\mathsf{RCI/CCI}}_{\Pi\circ\Gamma,\mathcal{CK}}(\mathcal{A})}$

$k \xleftarrow{\$} \mathrm{KG}()$
if RCI **then**
  $ck_0, ck_1 \xleftarrow{\$} \mathcal{CK}$ s.t. $|ck_0| = |ck_1|$
else if CCI **then**
  $ck_0, ck_1 \leftarrow \mathcal{A}^{\mathrm{E}_k(\cdot)}$ s.t. $|ck_0| = |ck_1|$
$b \xleftarrow{\$} \{0,1\}$
$(m'_0, m''_0, m'_1, m''_1) \leftarrow \mathcal{A}^{\mathrm{E}_k(\cdot\|ck_b\|\cdot),\mathrm{E}_k(\cdot)}(ck_0, ck_1)$
if $|\mathrm{Comp}(m'_0)| + |\mathrm{Comp}(m''_0)|$
  $\neq |\mathrm{Comp}(m'_1)| + |\mathrm{Comp}(m''_1)|$ **then**
  return 0
$c^* \leftarrow \mathrm{E}_k(m'_b\|ck_b\|m''_b)$
$b' \leftarrow \mathcal{A}^{\mathrm{E}_k(\cdot\|ck_b\|\cdot),\mathrm{E}_k(\cdot)}(c^*, ck_0, ck_1)$
if $b' = b$ **then**
  return 1
else
  return 0

Figure 3: Security experiments for cookie recovery (left) and random cookie indistinguishability and chosen cookie indistinguishability (right) attacks

2.4 Relations and separations between security notions

Cookie recovery, being a computational problem rather than a decisional problem, is a weaker security notion. Keeping CR as an initial step, the RCI and CCI notions gradual increase the security afforded to the cookie, as well as incorporating the message indistinguishability property as well.

We can establish the following relations between security notions for symmetric-key compression-encryption schemes:

$$\mathsf{CCI} \implies \mathsf{RCI} \implies \mathsf{CR} \ .$$

In other words, every scheme that provides chosen cookie indistinguishability provides random cookie indistinguishability, and so on. Moreover, these notions are distinct, and we can show separations between them:

$$\mathsf{CR} \not\Longrightarrow \mathsf{RCI} \not\Longrightarrow \mathsf{CCI} \ .$$

Additionally, we can connect our new notions with existing notions:

$$\mathsf{CCI} \implies \mathsf{ER\text{-}IND\text{-}CPA} \implies \mathsf{IND\text{-}CPA} \ .$$

(These last relations should be interpreted as discussed at the end of Section 2.2: a CCI-secure symmetric-key compression-encryption scheme, treated as an encryption scheme, is IND-CPA secure, and so on.) Appendix A contains proofs of these relations and counterexamples for the separations.

3 Reviewing proposed mitigation techniques

In this section we review couple of mitigation techniques against compression-based side-channel attacks such as CRIME and BREACH attacks and give reasons on why those mitigation techniques are not provably secure or not suitable in practice.

3.1 Disabling compression

CRIME attack occurs due to SSL/TLS compression which is little-used, while BREACH attack occurs due to HTTP compression which is widely-used. The TLS/SSL compression is taken place in the security layer which in between the application layer and the transport layer, whereas the HTTP compression is taken place in the application layer. Since SSL/TLS compression is little-used and disabled in most browsers by default, switching it off would not be a huge overhead for data transmission as long as HTTP compression is available. Thus, CRIME can be mitigated by switching-off the SSL/TLS compression. While the CRIME is mitigated by that way, switching off HTTP compression is not a practical solution to mitigate BREACH attack, even though it is the most effective mitigation technique (trivially IND-CPA-secure as long as the underlying encryption scheme is IND-CPA-secure). Because switching off both compressions will affect in transmission band-width.

3.2 Length hiding

Length hiding by adding random amount of padding to the responses seems to be a technique which can delay the attacker. The output length should always be smaller than the original input length, otherwise it will be more inefficient than disabling the HTTP compression. Hence, the size of the set of numbers where the random value is picked is smaller; a polynomial of the plaintext length. Weak random number generators may lead towards weak countermeasures, because with significant probability the adversary can guess the amount of random padding. Moreover, by observing the output length of the same input for number of times, the attacker can obtain sufficient amount of information to reveal the actual compressed size. It is not possible to prove the security of this mitigation technique, the only possibility is to obtain an information theoretic bound on the minimum amount of information that the adversary needs to reveal the secret. Besides this, due to the random amount of extra padding it is not possible to have a good compression.

3.3 Rate-limiting the requests

Rate-limiting the requests can prevent the attacker getting sufficient information for the attack. But this seems to be very difficult technique in practice; because a frequent legitimate customer of the website may find it is not possible to get the service at some times. Therefore this mitigation technique cannot be considered as a suitable one.

4 Separating secrets from user inputs

In this section we analyze an alternative mitigation technique against attacks that recover secrets from compressed data: separating secrets from user inputs. This mitigation technique is a generic mitigation technique against a whole class of compression-based side-channel attacks.

4.1 Basic idea

The basic idea of separating secrets from user inputs is: given an input, use a filter to separate all the secrets from the rest of the content, including user inputs. Then the rest of the content is compressed, while the secrets are kept uncompressed.

We model the filter for separating secrets from rest of the content using a reversible polynomial-time function f defined as $f : \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$. Given a filter f and a compression shcme Γ, the separating-secrets scheme $\mathrm{SS}_{f,\Gamma}$ is given in Figure 4.

$\underline{\mathrm{SS}_{f,\Gamma}.\mathrm{Comp}(pt)}$

1: $ck, pt_{ns} \leftarrow f(pt)$
2: $\widetilde{pt_{ns}} \leftarrow \Gamma.\mathrm{Comp}(pt_{ns})$
3: **return** $(ck, \widetilde{pt_{ns}})$

$\underline{\mathrm{SS}_{f,\Gamma}.\mathrm{Decomp}(ck, \widetilde{pt_{ns}})}$

1: $pt_{ns} \leftarrow \Gamma.\mathrm{Decomp}(\widetilde{pt_{ns}})$
2: $pt \leftarrow f^{-1}(ck, pt_{ns})$
3: **return** pt

Figure 4: Abstract separating-secrets compression scheme SS

4.2 CCI-security of basic separating-secrets technique

In this section we analyze the security of separating-secrets mitigation technique according to CCI notion. Let $\Pi = (\mathrm{KeyGen}, \mathrm{Enc}, \mathrm{Dec})$ be a IND-CPA-secure symmetric-key encryption scheme and $\mathrm{SS}_{f,\Gamma}$ be a separating-secrets compression scheme as given in Figure 4. We consider the security of the resulting symmetric-key compression-encryption scheme $\Pi \circ \mathrm{SS}_{f,\Gamma}$, showing that, if the filter f is effective at separating out secret cookies, then breaking chosen cookie indistinguishability of $\Pi \circ \mathrm{SS}_{f,\Gamma}$ is as hard as breaking the indistinguishability (IND-CPA) of the encryption scheme Π.

Theorem 4.1. *Let Π be a symmetric-key encryption scheme. Let $\mathcal{A}$ be any adversary against the chosen cookie indistinguishability security of the separating-secrets symmetric-key compression-encryption scheme $\Pi \circ \mathrm{SS}_{f,\Gamma}$. Assume that filter f separates all secret values from the plaintext. Then*

$$\mathrm{Adv}^{\mathsf{CCI}}_{\Pi \circ \mathrm{SS}_{f,\Gamma}, \mathcal{CK}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{IND\text{-}CPA}}_{\Pi}(\mathcal{B}) \ ,$$

where $\mathcal{B}$ is an algorithm, constructed using the adversary $\mathcal{A}$, against the IND-CPA *security of the underlying symmetric-key encryption scheme Π.*

In order to prove Theorem 4.1 we use the game hopping technique [21]; define a sequence of games and relate the adversary's advantage of distinguishing each game from the previous game to the advantage of breaking one of the underlying cryptographic primitive.

Proof. Assume that the adversary $\mathcal{A}$ can win the CCI challenge against the separating-secrets symmetric-key compression-encryption scheme $\Pi \circ \mathrm{SS}_{f,\Gamma}$ with non negligible advantage $\mathrm{Adv}^{\mathsf{CCI}}_{\Pi \circ \mathrm{SS}_{f,\Gamma},\mathcal{CK}}(\mathcal{A})$.

Game 1: This game is the original game defined in CCI security definition.
Game 2: Same as Game 1 with the following exception: For the challenge request $(m_0 = (m'_0, m''_0), m_1 = (m'_1, m''_1))$ and afterwards, for each encryption request, $m = (m', m'')$, of $\mathcal{A}$, Game 2 challenger randomly chooses ciphertexts c and sends to $\mathcal{A}$.

Analysis: The adversary's advantage of distinguishing each game from the previous game is investigated. $\mathrm{Succ}_{\mathrm{Game}\ x}(\mathcal{A})$ be the event that the adversary $\mathcal{A}$ wins Game x, $\mathrm{Adv}^{\mathrm{Game}\ x}_{\mathcal{A}}$ be the advantage of the adversary $\mathcal{A}$ of winning Game x.

Game 1: The original game. Hence,

$$\mathrm{Adv}^{\mathrm{Game}\ 1}_{\mathcal{A}} = \mathrm{Adv}^{\mathsf{CCI}}_{\Pi \circ \mathrm{SS}_{f,\Gamma},\mathcal{CK}}(\mathcal{A}). \tag{2}$$

Difference between Game 1 and Game 2: We introduce an algorithm $\mathcal{B}$ which is constructed using the adversary $\mathcal{A}$ as a subroutine. If $\mathcal{A}$ can distinguish the difference between Game 1 and Game 2, then $\mathcal{B}$ can be used against a IND-CPA challenger of Π. To answer the encryption requests of $\mathcal{A}$, the algorithm $\mathcal{B}$ uses a IND-CPA challenger in which the secret key is k. The algorithm $\mathcal{B}$ randomly chooses $\theta \xleftarrow{\$} \{0,1\}$. For each encryption request, $m = (m', m'')$, of $\mathcal{A}$, $\mathcal{B}$ simulates the encryption process $\mathrm{E}_k(m' \| ck_\theta \| m'')$ as, When $\mathcal{A}$ outputs (m_0, m_1), $\mathcal{B}$ sends $(M_0 = (ck_0 \| \widetilde{m_0}), M_1 = (ck_1 \| \widetilde{m_1}))$ to the IND-CPA challenger as

1: $(ck_\theta, \widetilde{m}) \leftarrow \mathrm{SS}_{f,\Gamma}.\mathrm{Comp}(m' \| ck_\theta \| m'')$
2: sends $(ck_\theta \| \widetilde{m})$ to the IND-CPA challenger
3: gets $c \leftarrow \mathrm{Enc}_k(ck_\theta \| \widetilde{m})$ from the IND-CPA challenger and sends c to $\mathcal{A}$.

the challenge request. The IND-CPA challenger randomly chooses $b \xleftarrow{\$} \{0,1\}$, computes $c^* \leftarrow \mathrm{Enc}_k(M_b)$ and sends c^* to $\mathcal{B}$ as the challenge. $\mathcal{B}$ sends c^* to $\mathcal{A}$ as the challenge. If $\theta = b$, Game 2 is identical to the Game 1, otherwise the simulation constructed by $\mathcal{B}$ is identical to Game 2. If $\mathcal{A}$ can distinguish the difference between Game 1 and Game 2, then $\mathcal{B}$ can be used against a IND-CPA challenger of the symmetric-key encryption scheme Π. Hence,

$$|\mathrm{Adv}^{\mathrm{Game}\ 1}_{\mathcal{A}} - \mathrm{Adv}^{\mathrm{Game}\ 2}_{\mathcal{A}}| \leq \mathrm{Adv}^{\mathsf{IND\text{-}CPA}}_{\Pi}(\mathcal{B}). \tag{3}$$

Semantic security of Game 2: Since the ciphertext c is chosen randomly, $\mathcal{A}$ does not have any advantage in Game 2. Hence,

$$\mathrm{Adv}^{\mathrm{Game}\ 2}_{\mathcal{A}} = 0. \tag{4}$$

Using equations 2–4 we find,

$$\mathrm{Adv}^{\mathsf{CCI}}_{\Pi \circ \mathrm{SS}_{f,\Gamma},\mathcal{CK}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{IND\text{-}CPA}}_{\Pi}(\mathcal{B}).$$

□

4.3 Separating secrets in HTML

Separating secrets from user inputs is a realistic mitigation technique against BREACH attack; because in application layer the fields which contain secrets (CSRF tokens, PII or any sensitive data) can be identified and separated from the HTTP response body. In order to implement separating secrets from user inputs in HTML we need to instantiate the abstract function f as f_{HTML}.

One possible method to separate secrets is to separate the content assigned to the `value` attribute of a HTML file. Among other uses, the `value` attribute defines the value of a specific field in a form. The HTML code segment of Figure 5 shows inclusion of a secret CSRF token as a hidden `input` field in a web form, which will appear in a HTML response body. By separating the content assigned into the `value` attribute it is possible to separate the CSRF token.

```
<form action="/money_transfer" method="post">
<input type="hidden" name="CSRFtoken"
       value="OWT4NmQlODE4ODRjN2Q1NTlhMmZlYWE...">
...
</form>
```

Figure 5: HTML code segment showing inclusion of CSRF token in a web form

Based on the HTML specification, the following (case-insensitive) regular expression can be used to separate out all data that is given in the `value` attribute of HTML elements:

```
value\s*=\s*"([^"]*)"|value\s*=\s*'([^']*)'|value\s*=\s*[^\s]*
```

Note that the above regular expression will also capture the `value` attribute of HTML elements other than hidden `input` elements, including elements such as `option`, that may not need to be treated as secret. Figure 6 explains a basic algorithm of the separating secrets in a HTML file.

$f_{\text{HTML}}(\textit{Input})$

```
1: P ← value\s*=\s*"([^"]*)"|value\s*=\s*'([^']*)'|value\s*=\s*[^\s]*
2: Line ← read line in Input
3: while Line != null do
4:    nonSecret ← replace in Line (P, "FILTERED-OUT")
5:    Matcher ← match in Line (P)
6:    while Matcher exists do
7:       Secret ← append the Matcher
8:    Line ← read line in Input
```

Figure 6: Separating secrets from user inputs in HTML

4.4 Experimental results

Table 1 shows the result of applying the above regular expression to separate secrets on the top 10 global websites of Alexa Top Sites. As most pages containing little data in `value` attributes, the total amount of space required to transmit the separated secrets + remaining data is not much more than when the full page is compressed.

Website	Original size	gzip full page		Separating secrets	
		Size	Compression ratio	Size	Compression ratio
Google.com	145599	41455	3.51	41502	3.51
Facebook.com	48226	13785	3.50	15863	3.04
Youtube.com	467928	41813	11.19	41893	11.17
Yahoo.com	444408	82572	5.38	83342	5.33
Baidu.com	74979	17519	4.28	17727	4.23
Wikipedia.org	48548	11217	4.33	11809	4.11
Twitter.com	57777	12520	4.61	16618	3.48
Qq.com	626297	124108	5.05	125747	4.98
Amazon.com	234609	54922	4.27	56278	4.17
Taobao.com	192068	23658	8.12	23898	8.04

Table 1: Compression performance, separating secrets technique

4.5 Discussion

The main drawback of the separating secrets mitigation technique is that the separation technique must be application-dependent.

The method we used to separate secrets in HTML—separating the data assigned to the `value` attribute—also separates some of the non-secrets as well. As an example, for a `button` element in a web page `value` defines the text on the button, which is a non-secret. Even so, this is not a problem

for leakage resilience security, as long as all the secrets are separated and kept uncompressed. A more optimum secret-separation technique which separates only the secrets will give better compression.

This filter also only captures certain types of secrets, such as CSRF tokens in hidden fields. It does not necessarily capture all values that might be considered sensitive. For example, should the titles of books in a search results page on an shopping site be considered secret? If so, an alternative separation filter would have to be developed. To provide complete certainty, secret separation would would require additional markup with which the developer clearly identifies which data should be treated as secret. All the values which are not separated will be compressed together with user inputs and other application data, and hence open to the compression-based side-channel.

5 Fixed-dictionary compression

The CRIME and BREACH attacks work because the dictionary used by the DEFLATE compression algorithm is adaptive: if the attacker injects a substring of the target secret into the plaintext nearby the secret itself, then the plaintext will compress more because of the repeated substring. Some early compression algorithms were non-adaptive, using a fixed dictionary mechanism. For example, Pike [18] used a fixed dictionary of 205 popular English words and a variable length coding mechanism to achieve a compression ratio of less than 4 bits per character for typical English text. Another recent algorithm, Smaz [20], similarly uses a fixed dictionary consisting of common digrams and trigrams from English and HTML source code, allowing it to compress even very short strings. Because the CRIME and BREACH attacks rely on the adaptivity of the compression dictionary, fixed-dictionary compression algorithms can offer resistance to such attacks while still providing some compression, albeit not as good as adaptive compression.

In this section, we investigate the use of fixed-dictionary compression in the context of encryption. We describe the basic idea of fixed-dictionary compression. We give a security notion, called *cookie recovery security*, that we show can be satisfied by compression-encryption schemes using fixed-dictionary compression. We present an example of a modern fixed-dictionary algorithm, SPAZ, and report on the compression ratios achieved by our algorithm.

5.1 Basic idea

In general, fixed-dictionary compression schemes work by advancing through the string x and looking to see if the current substring appears in the dictionary $\mathcal{D}$: if it does, then an encoding of the index of the substring is recorded, otherwise an encoding of the current substring is recorded. The compression scheme must specify the encoding rules in a way that unambiguously discriminates between the two cases.

An abstract version of a fixed-dictionary fixed-width compression algorithm FD is given in Figure 7. FD checks if the current substring of length w appears in the dictionary $\mathcal{D}$. If it does, it records the index of the substring in $\mathcal{D}$ and advances w characters. If it does not, it records the next ℓ characters directly, then advances. (Using $\ell > 1$ but $\ell < w$ may be more efficient when it comes to encodings.)

$\mathrm{FD}_{\mathcal{D},w,\ell}.\mathrm{Comp}(x)$

```
y ← empty string
i ← 1
while i ≤ |x| − w + 1 do
    if x_{i:w} ⊆ D then
        y ← y ‖ encoding of index of x_{i:w} in D
        i ← i + w
    else
        y ← y ‖ encoding of x_{i:ℓ}
        i ← i + ℓ
return y
```

$\mathrm{FD}_{\mathcal{D},w,\ell}.\mathrm{Decomp}(y)$

```
x ← empty string
i ← 1
while i ≤ |y| do
    if y_i is the encoding of an index then
        x ← x ‖ D_{y_i:w}
        i ← i + 1
    else
        x ← x ‖ decoding of y_{i:ℓ'}
        i ← i + ℓ'
return x
```

Figure 7: Abstract fixed-dictionary fixed-width compression scheme FD
Note the simplification that ℓ characters of x are encoded as ℓ' characters of y.

For example, if $\mathcal{D}$ =“`cookierecoveryattack`”, then $\mathrm{FD}_{\mathcal{D},4,2}.\mathrm{Comp}$(“`recover the cookie`”) yields `7ver_the_1ie`.

5.2 CR-security of basic fixed-dictionary technique

Let Π be a symmetric-key encryption scheme. Let $\mathcal{D}$ be a dictionary of length d and let $\mathrm{FD}_{\mathcal{D},w,\ell}$ be the abstract fixed-dictionary compression scheme shown in Figure 7.

Suppose the cookie space is binary strings of length 8λ, or equivalently byte strings of length λ: $\mathcal{CK} = \{\texttt{0x00}, \ldots, \texttt{0xFF}\}^{\lambda}$.

If Π is a secure encryption scheme, then, intuitively, the only way the adversary can learn information about the cookie from seeing ciphertexts $\mathrm{Enc}_k(\cdot\|ck\|\cdot)$ and $\mathrm{Enc}_k(\cdot)$ is from the length of the ciphertext: if some substring of ck appears in the dictionary $\mathcal{D}$, then ck will compress, and that length difference tells the adversary that the secret cookie is restricted to some subset of $\mathcal{CK}$ that matches $\mathcal{D}$.

The situation is slightly subtler in the full CR experiment: the attacker can provide strings a and b and get back $\mathrm{Enc}_k(\mathrm{Comp}(a\|ck\|b))$. If the last few bytes of a followed by the first few bytes of ck appear in $\mathcal{D}$, then the string will compress more. This allows the attacker to carry out a CRIME-like attack on the first few bytes of ck.

For example, let $w = 4$ and suppose that

$$\mathcal{D} = \texttt{1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz}\ .$$

If the the first byte of ck is a letter of the alphabet (in ASCII encoding), which happens with probability $52/256$, then the attacker can ask queries with $a = \texttt{890}$, $a = \texttt{90A}$, $a = \texttt{0AB}$, ... In exactly one case, the adversary's a prefix combined with the cookie's first letter will be in the dictionary, thereby telling the adversary the first byte of ck. For example, if the first byte of ck is `b`, then when the adversary queries $a = \texttt{YZa}$, the value that is compressed and then encrypted is $a\|ck\|b = \texttt{YZab}\ldots$, which starts with a dictionary substring.

While this allows the attacker to recover the first byte or two of the secret cookie with decent probability, it drops off exponentially; a similar argument applies to the last few bytes of the secret cookie. The final result below in Theorem 5.1 captures this issue, and only provides quantifiable security of the cookie length n is significantly bigger than the compression window w. (This is why we focus on a cookie *recovery* notion here, rather than a cookie *indistinguishability* notion).

Theorem 5.1. *Let Π be a symmetric-key encryption scheme. Let $\mathcal{D}$ be a dictionary of d words, each of length ℓ. Let w be positive integer. Let $\mathcal{CK} = \Omega^n$. Let $\mathcal{A}$ be any adversary against the cookie recovery security of the fixed-dictionary symmetric-key compression-encryption scheme $\Pi \circ \mathrm{FD}_{\mathcal{D},w,\ell}$. Then*

$$\mathrm{Adv}^{\mathsf{CR}}_{\Pi\circ\mathrm{FD}_{\mathcal{D},w,\ell}}(\mathcal{A}) \le \mathrm{Adv}^{\mathsf{IND\text{-}CPA}}_{\Pi}(\mathcal{B}) + 2^{-\Delta}\ ,$$

where $\mathcal{B}$ is an algorithm, constructed using adversary $\mathcal{A}$, against the IND-CPA *security of the underlying symmetric-key encryption scheme Π, and*

$$\Delta \ge \left(1 - d\left(1 - \left(1 - \frac{1}{|\Omega|^w}\right)^{n-3w+1}\right)\right) \cdot \log_2\left(|\Omega|^{n-2w} - |\Omega|^{n-2w} \cdot d\left(1 - \left(1 - \frac{1}{|\Omega|^w}\right)^{n-3w+1}\right)\right)\ .$$

For example, for cookies of $n = 16$ bytes, with a dictionary of $d = 4000$ words each of length $w = 4$, we have $\Delta \ge 63.999695$.

5.2.1 Probability bounds, no prefix/suffix.

We first compute the amount of information given to the adversary by knowing the length of the compressed cookie, without any adversarially chosen prefix or suffix. This can be computed by first calculating the amount of information given by knowing how many substrings of the cookie appear in the dictionary.

First we calculate the probability that a given string is a substring of a randomly chosen cookie. Note that in our setting, typically the character set $\Omega = \{\texttt{0x00}, \ldots, \texttt{0xFF}\}$ is the set of all bytes.

Lemma 5.2. *Let $x \in \Omega^w$ be a word, and let $ck \xleftarrow{\$} \Omega^n = \mathcal{CK}$ be a random string of n characters. Then*

$$\Pr(x \subseteq ck) \le 1 - \left(1 - \frac{1}{|\Omega|^w}\right)^{n-w+1}\ .$$

Proof.

$$\begin{aligned}\Pr(x \subseteq ck) &= 1 - \Pr(x \not\subseteq ck) \\ &= 1 - \Pr((x \neq ck_{1:w}) \wedge (x \neq ck_{2:w}) \wedge \cdots \wedge (x \neq ck_{n-w+1:w})) \\ &\leq 1 - \Pr(x \neq ck_{1:w}) \Pr(x \neq ck_{2:w}) \ldots \Pr(x \neq ck_{n-w+1:w}) \\ &= 1 - \left(1 - \frac{1}{|\Omega|^w}\right)^{n-w+1}\end{aligned}$$

□

We now compute that probability that one of set of given strings is a substring of a randomly chosen cookie:

Lemma 5.3. *Let $\mathcal{D} \subseteq \Omega^w$ with $|\mathcal{D}| = d$ be a dictionary of d words of w characters. Let $ck \xleftarrow{\$} \Omega^n = \mathcal{CK}$ be a random string of n characters. Then*

$$\Pr(\mathcal{D} \cap ck \neq \emptyset) \leq d\left(1 - \left(1 - \frac{1}{|\Omega|^w}\right)^{n-w+1}\right) \ .$$

Proof. Suppose $\mathcal{D} = \{x_1, x_2, \ldots, x_d\}$.

$$\begin{aligned}\Pr(\mathcal{D} \cap ck \neq \emptyset) &= \Pr((x_1 \subseteq ck) \vee (x_2 \subseteq ck) \vee \cdots \vee (x_d \subseteq ck)) \\ &\leq \sum_{i=1}^{d} \Pr(x_i \subseteq ck) \\ &\leq d\left(1 - \left(1 - \frac{1}{|\Omega|^w}\right)^{n-w+1}\right) \qquad \text{(by Lemma 5.2)}\end{aligned}$$

□

Recall the definition of conditional entropy: if X and Y are random variables, then

$$\begin{aligned}H(Y \mid X) &= \sum_{x \in \mathrm{supp}(X)} \Pr(X = x) H(Y \mid X = x) \\ &= -\sum_{x \in \mathrm{supp}(X)} \Pr(X = x) \sum_{y \in \mathrm{supp}(Y)} \Pr(Y = y \mid X = x) \log_2 \Pr(Y = y \mid X = x) \ .\end{aligned}$$

We now compute the amount of entropy about the cookie given knowledge about the number of substrings of the cookie that appear in the dictionary:

Lemma 5.4. *Fix $\mathcal{D}$. Let $\#\mathrm{SUB}(ck)$ denote the number of substrings of ck that appear in $\mathcal{D}$. Suppose CK is a uniform random variable on $\mathcal{CK}$. Then*

$$\begin{aligned}H(CK \mid \#\mathrm{SUB}(CK)) \geq &\left(1 - d\left(1 - \left(1 - \frac{1}{|\Omega|^w}\right)^{n-w+1}\right)\right) \\ &\cdot \log_2\left(|\mathcal{CK}| - |\mathcal{CK}| \cdot d\left(1 - \left(1 - \frac{1}{|\Omega|^w}\right)^{n-w+1}\right)\right) \ .\end{aligned}$$

Proof. Let $\#_s$ denote the number of cookies $ck \in \mathcal{CK}$ such that $\#\mathrm{SUB}(ck) = s$. First note that

$$\Pr(\#\mathrm{SUB}(CK) = s) = \frac{\#_s}{|\mathcal{CK}|} \ .$$

Additionally,

$$\Pr(CK = ck \mid \#\mathrm{SUB}(CK) = s) = \begin{cases} \frac{1}{\#_s}, & \text{if } |ck| = s \ , \\ 0, & \text{otherwise} \ . \end{cases}$$

Then

$$\sum_{ck \in \mathcal{CK}} \Pr(CK = ck \mid \#\mathrm{SUB}(CK) = s) = \#_s \cdot \frac{1}{\#_s} \log_2 \frac{1}{\#_s} = -\log_2 \#_s \ .$$

Substituting into the definition of conditional entropy, we have

$$H(CK \mid \#\mathrm{SUB}(CK)) = -\sum_{s\in\mathbb{N}} \Pr(\#\mathrm{SUB}(CK)=s) \sum_{ck\in\mathcal{CK}} \Pr(CK=ck \mid \#\mathrm{SUB}(CK)=s) \cdot \log_2 \Pr(CK=ck \mid \#\mathrm{SUB}(CK)=s)$$

$$= -\sum_{s\in\mathbb{N}} \frac{\#_s}{|\mathcal{CK}|}(-\log_2 \#_s)$$
$$= \frac{1}{|\mathcal{CK}|}\sum_{s\in\mathbb{N}} \#_s \log_2 \#_s \ .$$

Let $\#_{\geq 1}$ denote the number of cookies $ck \in \mathcal{CK}$ such that $\#\mathrm{SUB}(ck) \geq 1$. Then

$$\Pr(\#\mathrm{SUB}(CK)\geq 1) = \Pr(\mathcal{D}\cap CK \neq \emptyset) = \frac{\#_{\geq 1}}{|\mathcal{CK}|} \qquad \text{(by definition of } \#_{\geq 1})$$
$$\leq d\left(1-\left(1-\frac{1}{|\Omega|^w}\right)^{n-w+1}\right) \qquad \text{(by Lemma 5.3)}$$

Thus, the number of cookies where at least 1 substring appears in the dictionary is at least

$$\#_{\geq 1} \leq |\mathcal{CK}|\cdot d\left(1-\left(1-\frac{1}{|\Omega|^w}\right)^{n-w+1}\right) \ .$$

Consequently, the number of cookies where no substring appears in the dictionary is at most

$$\#_0 = |\mathcal{CK}| - \#_{\geq 1} \geq |\mathcal{CK}| - |\mathcal{CK}| d\left(1-\left(1-\frac{1}{|\Omega|^w}\right)^{n-w+1}\right) \ .$$

Finally,

$$H(CK \mid \#\mathrm{SUB}(CK)) = \frac{1}{|\mathcal{CK}|}\sum_{s\in\mathbb{N}} \#_s \log_2 \#_s$$
$$\geq \frac{1}{|\mathcal{CK}|}\#_0 \log_2 \#_0$$
$$\geq \left(1-d\left(1-\left(1-\frac{1}{|\Omega|^w}\right)^{n-w+1}\right)\right) \cdot \log_2\left(|\mathcal{CK}| - |\mathcal{CK}|\cdot d\left(1-\left(1-\frac{1}{|\Omega|^w}\right)^{n-w+1}\right)\right) \ .$$

□

For example, if we have 16-byte cookies ($\mathcal{CK} = \{\texttt{0x00},\ldots,\texttt{0xFF}\}^{16}$), and the dictionary $\mathcal{D}$ is a set of $d = 4096$ words of length $w = 4$ bytes, then

$$H(CK \mid \#\mathrm{SUB}(CK)) \geq 127.998395 \ .$$

Concluding our analysis of the information learned given to the adversary without any adversarially chosen prefix or suffix, we give a bound on the amount of entropy about the cookie given the length of the compressed cookie:

Lemma 5.5. *Fix $\mathcal{D}$ with d words of length w over character set Ω. Let $\mathrm{COMPLEN}(ck) = |\mathrm{FD}_{\mathcal{D},w,\ell}.\mathrm{Comp}(ck)|$ denote the length of a cookie ck compressed with dictionary $\mathcal{D}$. Suppose CK is a uniform random variable on $\mathcal{CK}$. Then*

$$H(CK \mid \mathrm{COMPLEN}(CK)) \geq H(CK \mid \#\mathrm{SUB}(CK))$$
$$\geq \left(1-d\left(1-\left(1-\frac{1}{|\Omega|^w}\right)^{n-w+1}\right)\right) \cdot \log_2\left(|\mathcal{CK}| - |\mathcal{CK}|\cdot d\left(1-\left(1-\frac{1}{|\Omega|^w}\right)^{n-w+1}\right)\right) \ .$$

Lemma 5.5 is an immediate consequence of the data processing inequality and Lemma 5.4.

5.2.2 Probability bounds, prefix/suffix.

Suppose CK is a uniform random variable on $\mathcal{CK} = \Omega^n$. We know that $H(CK) = n\log_2(|\Omega|)$. Trivially, $H(CK \mid CK_1) = (n-1)\log_2(|\Omega|)$, where CK_1 is the first character of CK. Similarly, $H(CK \mid CK_{1:a}) = (n-a)\log_2(|\Omega|)$ and finally $H(CK \mid CK_{1:a}, CK_{n-b:b}) = (n-a-b)\log_2(|\Omega|)$.

Consider the following CRIME-like attack on the beginning of the cookie. Let $\mathcal{D}$ be a dictionary with d words of length w over character set Ω. Let $ck \in \Omega^n$. Let $O(\cdot)$ be an oracle that, upon input a of length $w-m$, with $1 \le m \le w-1$, returns 1 if and only if $a \| ck_{1:m} \in \mathcal{D}$.

The CRIME-like attack works as follows:

1. For each $x \in \mathcal{D}$, query $x_{1:w-1}$ to the oracle. If a query for $x_{1:w-1}$ returns 1, then it is known that $ck_{1:1} \in Z_1 = \{z : x_{1:w-1} \| z \in \mathcal{D}\}$. If no query returns 1, then return $\emptyset$.

2. For $m = 2, \ldots, w-1$: For each $x \in \mathcal{D}$ such that $x_{w-m} \in Z_{m-1}$, query $x_{1:w-m}$ to the oracle. If a query for $x_{1:w-m}$ returns 1, then it is known that $ck_{1:m} \in Z_m = \{z_1 z_2 \ldots z_m : x_{1:w-m} \| z_1 z_2 \ldots z_m \in \mathcal{D}\}$. If no query returns 1, then return $Z_1, \ldots, Z_{m-1}$.

3. Return $Z_1, \ldots, Z_{w-1}$.

A corresponding attack on the suffix is obvious.

Let $\text{CRIMEpre}(ck)$ denote the output obtained from running the above prefix CRIME attacks on ck, $\text{CRIMEsuf}(ck)$ denote the output from the corresponding suffix attack. Let $\text{CRIME}(ck) = (\text{CRIMEpre}(ck), \text{CRIMEsuf}(ck))$.

Noting that in the best case the CRIME attack allows the attacker to learn the first $w-1$ and the last $w-1$ characters of the cookie, some trivial lower bounds are:

$$H(CK_{1:w-1} \mid \text{CRIME}(CK)) \ge 0$$
$$H(CK_{n-w+1:w-1} \mid \text{CRIME}(CK)) \ge 0$$

However, the CRIME attack provides no information about the remaining characters, so $I(CK_{1:w-1}, CK_{w:n-w+1}) = 0$ and $I(CK_{1:n-w+1}, CK_{n-w+1:w-1}) = 0$, and thus

$$H(CK_{w:n-w+2} \mid \text{CRIME}(CK), \text{COMPLEN}(CK)) = H(CK_{w:n-w+2} \mid \text{COMPLEN}(CK)) \ .$$

Finally, we have that

$$\begin{aligned} &H(CK \mid \text{CRIME}(CK), \text{COMPLEN}(CK)) \\ &\quad \ge H(CK_{1:w-1} \mid \text{CRIMEpre}(CK)) + H(CK_{w:n-w+2} \mid \text{COMPLEN}(CK)) \\ &\qquad + H(CK_{n-w+1:w-1} \mid \text{CRIMEsuf}(CK)) \\ &\quad \ge 0 + H(CK_{w:n-w+2} \mid \text{COMPLEN}(CK)) + 0 \end{aligned}$$

and we can obtain a lower bound on $H(CK_{w:n-w} \mid \text{COMPLEN}(CK))$ using Lemma 5.5.

5.3 Experimental results

Table 2 shows the result of applying a fixed-dictionary based compression algorithm on the top 10 global websites of Alexa Top Sites. The 4000-byte dictionary was built from the most common 8-, 16-, and 32-character substrings of the pages. The compression algorithm was based in part on the Smaz [20] algorithm, and was adapted slightly from Figure 7 to allow for variable-length words to be matched.

5.4 Discussion

The main drawback of the fixed dictionary mitigation technique is that it achieves relatively poor—albeit non-zero—compression compared with adaptive compression techniques. However, it does not rely on application-dependent or heuristic techniques for deparating secrets.

Website	Original size	gzip full page		Fixed dictionary	
		Size	Compression ratio	Size	Compression ratio
Google.com	145599	41455	3.51	117794	1.23
Facebook.com	48226	13785	3.50	35036	1.37
Youtube.com	467928	41813	11.19	181676	2.58
Yahoo.com	444408	82572	5.38	318386	1.40
Baidu.com	74979	17519	4.28	55950	1.34
Wikipedia.org	48548	11217	4.33	38406	1.26
Twitter.com	57777	12520	4.61	39712	1.46
Qq.com	626297	124108	5.05	519830	1.21
Amazon.com	234609	54922	4.27	150924	1.55
Taobao.com	192068	23658	8.12	93410	2.06

Table 2: Compression performance, fixed-dictionary technique

6 Conclusion

In this paper we introduced theoretical models to analyze compression-based side-channel attacks on high-value secrets embedded inside messages: the notions of cookie recovery (CR) security, random cookie indistinguishability (RCI), and chosen cookie indistinguishability (CCI).

The simple, but relatively weak, CR security notion which allows adaptive access to an oracle what encrypts chosen plaintexts alongside a target cookie, is sufficient to address real-world compression-based side-channel attacks such as CRIME and BREACH that aim to recover the target secret. The CCI security notion address stronger situations where the adversary has to decide which of two secrets of the adversary's choice was embedded in the encrypted plaintext, even given adaptive access to an oracle that encrypts plaintexts of his choosing with the target secret embedded.

The most secure countermeasure to compression-based side-channel attacks remains to disable compression. As implementers seem loathe to do so—indeed, compression is even more heavily embedded in current drafts of HTTP version 2 [4, §10.6] than it was in previous versions—techniques for safely compression data that may be partially adversarially controlled are of significant importance. While compression inherently leaks information about redundancy in plaintext, some compression techniques, such as the separating secrets and fixed dictionary approaches treated in this paper, provide some resistance to previous compression-based attacks like CRIME and BREACH. Further cryptographic study of compression seems like a worthwhile research direction.

References

[1] A. Akavia, S. Goldwasser, and V. Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In O. Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 474–495. Springer, Mar. 2009.

[2] J. Alwen, Y. Dodis, and D. Wichs. Leakage-resilient public-key cryptography in the bounded-retrieval model. In S. Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 36–54. Springer, Aug. 2009.

[3] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In H. Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 26–45. Springer, Aug. 1998.

[4] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol version 2, July 2014. Internet-Draft. http://tools.ietf.org/html/draft-ietf-httpbis-http2-14.

[5] D. J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.

[6] D. Brumley and D. Boneh. Remote timing attacks are practical. In *USENIX Security Symposium*, pages 1–14, 2003.

[7] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[8] S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *49th FOCS*, pages 293–302. IEEE Computer Society Press, Oct. 2008.

[9] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230 (Proposed Standard), June 2014.

[10] Y. Gluck, N. Harris, and A. Prado. SSL, gone in 30 seconds: A BREACH beyond CRIME. In *Black Hat USA 2013*, August 2013.

[11] S. Halevi and H. Lin. After-the-fact leakage in public-key encryption. In Y. Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 107–124. Springer, Mar. 2011.

[12] M. Hutter, S. Mangard, and M. Feldhofer. Power and EM attacks on passive 13.56 mhz RFID devices. In P. Paillier and I. Verbauwhede, editors, *CHES 2007*, volume 4727 of *LNCS*, pages 320–333. Springer, Sept. 2007.

[13] J. Kelley and R. Tamassia. Secure compression: Theory & practice. Cryptology ePrint Archive, Report 2014/113, 2014. http://eprint.iacr.org/2014/113.

[14] J. Kelsey. Compression and information leakage of plaintext. In J. Daemen and V. Rijmen, editors, *FSE 2002*, volume 2365 of *LNCS*, pages 263–276. Springer, Feb. 2002.

[15] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, Aug. 1996.

[16] T. Messerges, E. Dabbish, and R. Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Transactions on Computers*, 51:541–552, 2002.

[17] M. Naor and G. Segev. Public-key cryptosystems resilient to key leakage. In S. Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 18–35. Springer, Aug. 2009.

[18] J. Pike. Text compression using a 4 bit coding scheme. *The Computer Journal*, 24(4):324–330, September 1980.

[19] J. Rizzo and T. Duong. The CRIME attack, 2012. Presented at ekoparty '12. http://goo.gl/mlw1X1.

[20] S. Sanfilippo. Smaz: Small strings compression library, April 2009. https://github.com/antirez/smaz.

[21] V. Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. http://eprint.iacr.org/2004/332.

[22] Trustworthy Internet Movement. SSL Pulse, September 2014. https://www.trustworthyinternet.org/ssl-pulse/.

A Relations and separations between security notions

Note that $\Pi = (\mathrm{KeyGen}, \mathrm{Enc}, \mathrm{Dec})$ be a symmetric-key encryption scheme and $\Gamma = (\mathrm{Comp}, \mathrm{Decomp})$ be a compression scheme. The structure of the resulting symmetric-key compression-encryption scheme is $\Pi \circ \Gamma = (\mathrm{KG} = \Pi.\mathrm{KeyGen}, \mathrm{E} = \Pi.\mathrm{Enc} \circ \Gamma.\mathrm{Comp}, \mathrm{D} = \Gamma.\mathrm{Decomp} \circ \Pi.\mathrm{Enc})$.

Theorem A.1. RCI*-secure symmetric-key compression-encryption scheme* $\Rightarrow$ CR*-secure symmetric-key compression-encryption scheme.*

Proof. Let $\mathcal{A}$ be a PPT adversary against the CR security challenge of a symmetric-key compression-encryption scheme $\Pi \circ \Gamma = (\mathrm{KG}, \mathrm{E}, \mathrm{D})$. Assume that $\mathcal{A}$ can win the CR security challenge with non-negligible probability. We construct an algorithm $\mathcal{B}$ against the RCI security challenge of $\Pi \circ \Gamma$ using $\mathcal{A}$ as a subroutine. The algorithm $\mathcal{B}$ simulates the view of CR challenger to $\mathcal{A}$ and constructs the RCI adversary against the RCI challenger. The simulation of $\mathcal{B}$ is explained in Figure 8.

The simulation in Figure 8 illustrates that if $\mathcal{A}$ can win the CR challenge of $\Pi \circ \Gamma$ with non-negligible probability, then $\mathcal{B}$ can win the RCI challenge of $\Pi \circ \Gamma$ with non-negligible advantage. Hence, if $\mathcal{B}$ can not win the RCI challenge of $\Pi \circ \Gamma$ with non-negligible advantage, then $\mathcal{A}$ can not win CR challenge of $\Pi \circ \Gamma$ with non-negligible probability. Thus, RCI-secure symmetric-key compression-encryption scheme $\Rightarrow$ CR-secure symmetric-key compression-encryption scheme. □

Theorem A.2. CR*-secure symmetric-key compression-encryption scheme* $\not\Rightarrow$ RCI*-secure symmetric-key compression-encryption scheme.*

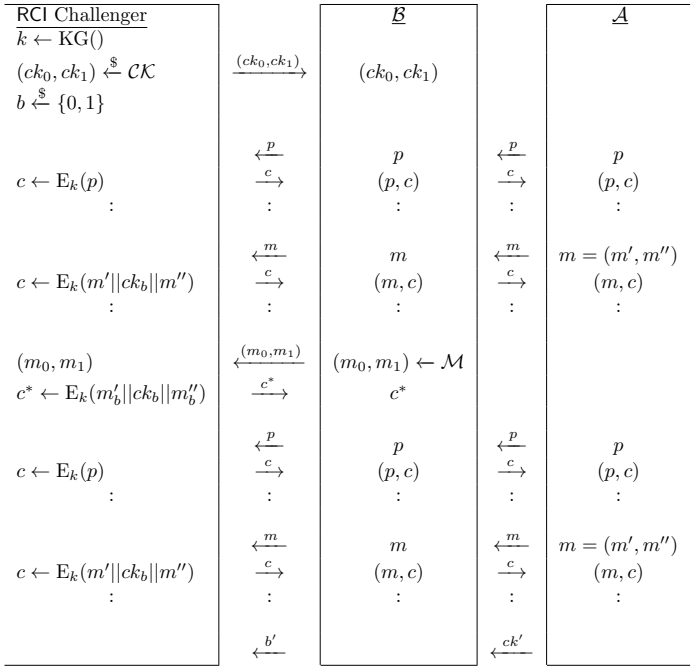

Figure 8: The CR challenge simulation of $\mathcal{B}$

Proof. In order to prove this theorem we use a proof strategy used in Bellare et al. [3]. We assume that a symmetric-key compression-encryption scheme $\Pi \circ \Gamma = (\mathrm{KG}, \mathrm{E}, \mathrm{D})$ is CR-secure. We now modify $\Pi \circ \Gamma$ to a new symmetric-key encryption scheme $(\Pi \circ \Gamma)' = (\mathrm{KG}, \mathrm{E}', \mathrm{D}')$ which is also CR-secure, but not RCI-secure. This will prove the Theorem A.2.

Let p^t be the first t bits of the plaintext p where $|p| > t$ and H be a second preimage resistant hash function where the input size is t. A ciphertext of $(\Pi \circ \Gamma)'$ is a pair of components, $(c_1||c_2)$, where c_1 consists encryption of the plaintext and c_2 consists $\mathrm{H}(p^t)$. The decryption ignores the second component of the ciphertext and simply outputs the decryption of the first component. The new symmetric-key encryption scheme $(\Pi \circ \Gamma)'$ is described in Figure 9.

$k \leftarrow \mathrm{KG}()$	$\mathrm{E}'_k(p)\{$	$\mathrm{D}'_k(c_1\|\|c_2)\{$
	$c_1 \leftarrow \mathrm{E}_k(p); c_2 \leftarrow \mathrm{H}(p^t)$	$p \leftarrow \mathrm{D}_k(c_1)$
	return $(c_1\|\|c_2)\}$	return $p\}$

Figure 9: Encryption Scheme $(\Pi \circ \Gamma)'$

Claim A.3. *The symmetric-key encryption scheme* $(\Pi \circ \Gamma)' = (\mathrm{KG}, \mathrm{E}', \mathrm{D}')$ *is* CR*-secure.*

If $\Pi \circ \Gamma$ is CR-secure and H is a second preimage resistant hash function, upon seeing c_2 the adversary gain no information about the first t bits of p. Thus, if the underlying symmetric-key encryption scheme is CR-secure and the hash function H is second preimage resistant, the symmetric-key encryption scheme $(\Pi \circ \Gamma)'$ is CR-secure.

Claim A.4. *The symmetric-key encryption scheme* $(\Pi \circ \Gamma)' = (\mathrm{KG}, \mathrm{E}', \mathrm{D}')$ *is not* RCI*-secure.*

If the adversary is given two t-bit strings such that one bit string is the first t bits of the plaintext p and the other bit string is a random t-bit string, upon seeing c_2 the adversary can obviously find the first t-bits of the plaintext by checking H of which bit-string matches with c_2. Thus, although the underlying symmetric-key encryption scheme is CR-secure, the symmetric-key encryption scheme $(\Pi \circ \Gamma)'$ is not CR-secure. Thus, CR-secure symmetric-key compression-encryption scheme $\not\Rightarrow$ RCI-secure symmetric-key compression-encryption scheme. □

Theorem A.5. CCI*-secure symmetric-key compression-encryption scheme* $\Rightarrow$ RCI*-secure symmetric-key compression-encryption scheme.*

Proof. Let $\mathcal{A}$ be a PPT adversary against RCI security of a symmetric-key compression-encryption scheme $\Pi \circ \Gamma = (\mathrm{KG}, \mathrm{E}, \mathrm{D})$. Assume that $\mathcal{A}$ can win the RCI security game with non-negligible probability. We construct an algorithm $\mathcal{B}$ against CCI security of $\Pi \circ \Gamma$ using $\mathcal{A}$ as a subroutine. The algorithm $\mathcal{B}$

simulates the view of RCI challenger to $\mathcal{A}$ and constructs the CCI adversary against the CCI challenger. The simulation of $\mathcal{B}$ is explained in Figure 10.

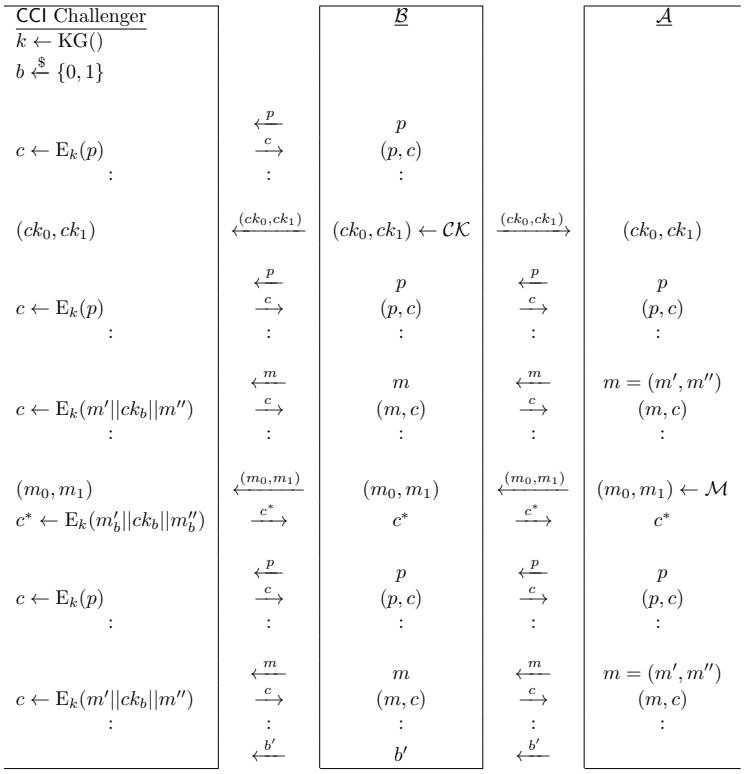

Figure 10: The RCI challenge simulation of $\mathcal{B}$

The simulation in Figure 10 illustrates that if $\mathcal{A}$ can win the RCI challenge of $\Pi \circ \Gamma$ with non-negligible advantage, then $\mathcal{B}$ can win the CCI challenge of $\Pi \circ \Gamma$ with non-negligible advantage. Hence, if $\mathcal{B}$ can not win the CCI challenge of $\Pi \circ \Gamma$ with non-negligible advantage, then $\mathcal{A}$ can not win RCI challenge of $\Pi \circ \Gamma$ with non-negligible advantage. Thus, CCI-secure symmetric-key compression-encryption scheme $\Rightarrow$ RCI-secure symmetric-key compression-encryption scheme. □

Theorem A.6. RCI*-secure symmetric-key compression-encryption scheme* $\not\Rightarrow$ CCI*-secure symmetric-key compression-encryption scheme.*

Proof. We assume that a symmetric-key compression-encryption scheme $\Pi \circ \Gamma = (\mathrm{KG}, \mathrm{E}, \mathrm{D})$ is RCI-secure. We now modify $\Pi \circ \Gamma$ to a new symmetric-key encryption scheme $(\Pi \circ \Gamma)'' = (\mathrm{KG}, \mathrm{E}'', \mathrm{D}'')$ which is also RCI-secure, but not CCI-secure. This will prove the Theorem A.6.

Let p^t be the first t bits of the plaintext p where $|p| > t$ and f be a point function as defined follows where the input size is t.

$$f(x) = \begin{cases} 0 & \text{if } x = \text{public hard-coded value} \\ 1 & \text{otherwise} \end{cases}$$

A ciphertext of $(\Pi \circ \Gamma)''$ is a pair of components, $(c_1||c_2)$, where c_1 consists encryption of the message and c_2 consists $f(p^t)$. The decryption ignores the second component of the ciphertext and simply outputs the decryption of the first component. The new symmetric-key encryption scheme $(\Pi \circ \Gamma)'' = (\mathrm{KG}, \mathrm{E}'', \mathrm{D}'')$ is described in Figure 11.

$k \leftarrow \mathrm{KG}()$	$\mathrm{E}''_k(p)\{$ $c_1 \leftarrow \mathrm{E}_k(p); c_2 \leftarrow f(p)$ return $(c_1\|\|c_2)\}$	$\mathrm{D}''_k(p)\{$ $p \leftarrow \mathrm{D}_k(c_1)$ return $p\}$

Figure 11: Encryption Scheme $(\Pi \circ \Gamma)''$

Claim A.7. *The symmetric-key encryption scheme* $\Pi'' = (\mathrm{KG}, \mathrm{E}'', \mathrm{D}'')$ *is* RCI*-secure.*

If $\Pi \circ \Gamma$ is RCI-secure and f is a point function, upon seeing c_2 the adversary gain no information about the first t bits of p, unless it equals to the public hard-coded value. Thus, if the underlying symmetric-key encryption scheme is RCI-secure and the function f is a point function, the symmetric-key encryption scheme $(\Pi \circ \Gamma)''$ is RCI-secure.

Claim A.8. *The symmetric-key encryption scheme* $(\Pi \circ \Gamma)'' = (\mathrm{KG}, \mathrm{E}'', \mathrm{D}'')$ *is not* CCI*-secure.*

If the adversary chooses two plaintexts such that in an one plaintext the first t bits equals to the public hard-coded value, upon seeing c_2 adversary can easily find whether the first t bits of the plaintext equals to the hard-coded value or not. Thus, although the underlying symmetric-key encryption scheme is RCI-secure, the symmetric-key encryption scheme $(\Pi \circ \Gamma)''$ is not CCI-secure. Thus, RCI-secure symmetric-key compression-encryption scheme $\not\Rightarrow$ CCI-secure symmetric-key compression-encryption scheme. □

Theorem A.9. CCI*-secure symmetric-key compression-encryption scheme* ⇒ IND-CPA*-secure symmetric-key compression-encryption scheme.*

Proof. Let $\mathcal{A}$ be a PPT adversary against IND-CPA security of a symmetric-key compression-encryption scheme $\Pi \circ \Gamma = (\mathrm{KG}, \mathrm{E}, \mathrm{D})$. Assume that $\mathcal{A}$ can win the IND-CPA security game with non-negligible probability. We construct an algorithm $\mathcal{B}$ against CCI security of $\Pi \circ \Gamma$ using $\mathcal{A}$ as a subroutine. The algorithm $\mathcal{B}$ simulates the view of IND-CPA challenger to $\mathcal{A}$ and constructs the CCI adversary against the CCI challenger. The simulation of $\mathcal{B}$ is explained in Figure 12.

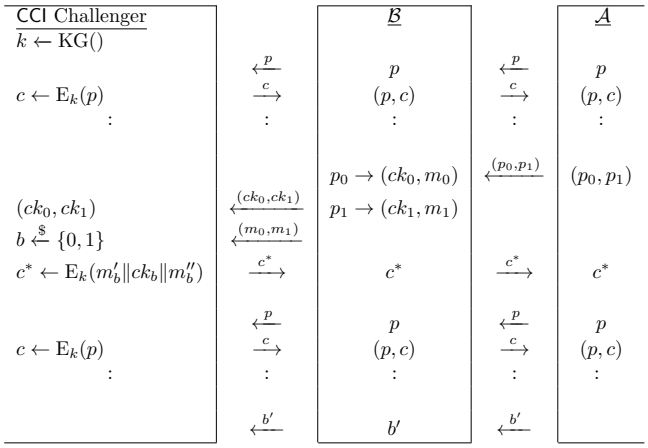

Figure 12: Simulation of $\mathcal{B}$ as the IND-CPA challenger

The simulation in Figure 12 illustrates that if $\mathcal{A}$ can win the IND-CPA challenge of $\Pi \circ \Gamma$ with non-negligible advantage, then $\mathcal{B}$ can win the CCI challenge of $\Pi \circ \Gamma$ with non-negligible advantage. Hence, if $\mathcal{B}$ can not win the IND-CPA challenge of $\Pi \circ \Gamma$ with non-negligible advantage, then $\mathcal{A}$ can not win CCI challenge of $\Pi \circ \Gamma$ with non-negligible advantage. Thus, CCI security ⇒ IND-CPA security. □

Theorem A.10. CCI*-secure symmetric-key compression-encryption scheme* ⇒ ER-IND-CPA*-secure symmetric-key compression-encryption scheme.*

Proof. Let $\mathcal{A}$ be a PPT adversary against ER-IND-CPA security of a symmetric-key compression-encryption scheme $\Pi \circ \Gamma = (\mathrm{KG}, \mathrm{E}, \mathrm{D})$. Assume that $\mathcal{A}$ can win the ER-IND-CPA security game with non-negligible probability. We construct an algorithm $\mathcal{B}$ against CCI security of $\Pi \circ \Gamma$ using $\mathcal{A}$ as a subroutine. The algorithm $\mathcal{B}$ simulates the view of ER-IND-CPA challenger to $\mathcal{A}$ and constructs the CCI adversary against the CCI challenger. The simulation of $\mathcal{B}$ is explained in Figure 13.

The simulation in Figure 13 illustrates that if $\mathcal{A}$ can win the ER-IND-CPA challenge of $\Pi \circ \Gamma$ with non-negligible advantage, then $\mathcal{B}$ can win the CCI challenge of $\Pi \circ \Gamma$ with non-negligible advantage. Hence, if $\mathcal{B}$ can not win the ER-IND-CPA challenge of $\Pi \circ \Gamma$ with non-negligible advantage, then $\mathcal{A}$ can not win CCI challenge of $\Pi \circ \Gamma$ with non-negligible advantage. Thus, CCI security ⇒ ER-IND-CPA security. □

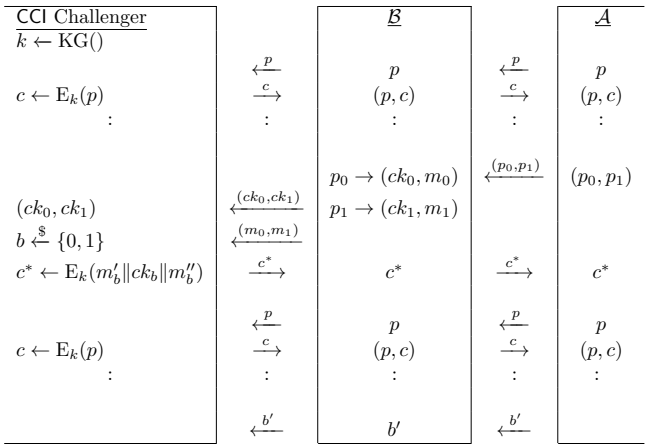

Figure 13: Simulation of $\mathcal{B}$ as the ER-IND-CPA challenger

B Source codes of fixed-dictionary mitigation technique

B.1 Source code of constructing a fixed-dictionary

```
1  import java.nio.file.*;
2  import java.util.*;
3  public class Freqs {
4          public static final int RECORD_SIZE = 2;
5          public static void main(String args[]) throws Exception {
6          if (args.length < 3) {
7                  System.err.println("Usage: java Freqs dictSize wordLengths filenames -> output.dict");
8                  System.err.println("- dictSize: the target size of the output dictionary in bytes; for Spaz this should be 4000");
9                  System.err.println("- wordLengths: comma-separated list of word lengths to count frequencies of; e.g., 8,15,20");
10                 System.err.println("- filenames: list of filenames to build dictionary from");
11                 return;
12         }
13         int dictSize = Integer.parseInt(args[0]);
14         String wordlengths[] = args[1].split(",");
15         HashMap<String, Integer> worths = new HashMap<String, Integer>();
16         for (int f = 2; f < args.length; f++) {
17                 String filename = args[f];
18                 String src = new String(Files.readAllBytes(Paths.get(filename)));
19                 for (String wordlength : wordlengths) {
20                         calcWorth(src, worths, Integer.parseInt(wordlength));
21                 }
22         }
23         printBest(worths, dictSize);
24 }
25
26 static void calcWorth(String src, Map<String, Integer> freqs, int seqlen) {
27         for (int i = 0; i < src.length() - seqlen; i++) {
28                 String s = src.substring(i, i + seqlen);
29                 if (freqs.containsKey(s)) {
30                         freqs.put(s, freqs.get(s) + seqlen - RECORD_SIZE);
31                 } else {
32                         freqs.put(s, seqlen - RECORD_SIZE);
33                         }
34         }
35 }
36
37 static void printBest(Map<String, Integer> freqs, int dictSize) {
38         Map<String, Integer> freqs_sorted = Freqs.sortByValue(freqs);
39         Vector<String> best = new Vector<String>(dictSize);
40         Iterator<Map.Entry<String, Integer>> it = freqs_sorted.entrySet().iterator();
41         int currDictSize = 0;
42         while (it.hasNext() && (currDictSize < dictSize)) {
43                 Map.Entry<String, Integer> pair = it.next();
```

```
                boolean useit = true;
                for (String v : best) {
                        if (longestSubstr(pair.getKey(), v) > 5) {
                                useit = false;
                                break;
                        }
                }
                if (useit) {
                        best.add(pair.getKey());
                        System.out.print(pair.getKey());
                        currDictSize += pair.getKey().length();
                }
        }
        System.out.println();
}

public static <K, V extends Comparable<? super V>> Map<K, V> sortByValue(Map<K, V> map) {
        List<Map.Entry<K, V>> list = new LinkedList<Map.Entry<K, V>>(map.entrySet());
        Collections.sort(list, new Comparator<Map.Entry<K, V>>() {
                public int compare(Map.Entry<K, V> o1, Map.Entry<K, V> o2) {
                        return -(o1.getValue()).compareTo(o2.getValue());
                }
        });
        Map<K, V> result = new LinkedHashMap<K, V>();
        for (Map.Entry<K, V> entry : list) {
                result.put(entry.getKey(), entry.getValue());
        }
        return result;
}

public static int longestSubstr(String first, String second) {
        if (first == null || second == null || first.length() == 0 || second.length() == 0) {
                return 0;
        }
        int maxLen = 0;
        int fl = first.length();
        int sl = second.length();
        int[][] table = new int[fl + 1][sl + 1];
        for (int s = 0; s <= sl; s++)
                table[0][s] = 0;
        for (int f = 0; f <= fl; f++)
                table[f][0] = 0;
        for (int i = 1; i <= fl; i++) {
                for (int j = 1; j <= sl; j++) {
                        if (first.charAt(i - 1) == second.charAt(j - 1)) {
                                if (i == 1 || j == 1) {
                                        table[i][j] = 1;
                                } else {
                                        table[i][j] = table[i - 1][j - 1] + 1;
                                }
                                if (table[i][j] > maxLen) {
                                        maxLen = table[i][j];
                                }
                        }
                }
        }
                return maxLen;
        }
}
```

B.2 Source code of fixed-dictionary experiment

```
import java.nio.file.*;
public class Spaz2 {
        public static final String dictionary = "dictionary";
        public static String decode(byte[] c, String dictionary) throws Exception {
                byte bytes[] = new byte[8*c.length];
                int bindex = 0;
                int cindex = 0;
```

```
8                   while (cindex < c.length) {
9                           byte b = c[cindex];
10                          if (b >> 6 == 0) {
11                                  if (cindex + 1 >= c.length) {
12                                          throw new Exception("Need 2 byte record");
13                                  }
14                          byte a = c[cindex];
15                          a &= 0x3F;
16                          a <<= 1;
17                          byte a2 = c[cindex+1];
18                          a2 >>>= 7;
19                          a2 &= 0x01;
20                          a |= a2;
21                          bytes[bindex] = a;
22                          a = c[cindex+1];
23                          a &= 0x7F;
24                          bytes[bindex+1] = a;
25                          cindex += 2;
26                          bindex += 2;
27                          } else if ((b & 0x80) > 0) {
28                                  if (cindex + 1 >= c.length) {
29                                          throw new Exception("Need 2 byte record");
30                                  }
31                          int j = c[cindex+1] & 0x07;
32                          int len = 2 * j + 4;
33                          int index = c[cindex] & 0x7F;
34                          index <<= 5;
35                          int index2 = c[cindex+1] & 0xF8;
36                          index2 >>= 3;
37                          index2 &= 0x1F;
38                          index |= index2;
39                          String dictlookup = dictionary.substring(index, index + len);
40                          byte dictlookupbytes[] = dictlookup.getBytes("UTF-8");
41                          System.arraycopy(dictlookupbytes, 0, bytes, bindex,
                                dictlookupbytes.length);
42                          cindex += 2;
43                          bindex += dictlookupbytes.length;
44                          } else if ((b & 0x40) > 0) {
45                                  if (cindex + 1 >= c.length) {
46                                          throw new Exception("Need 2 byte record");
47                                  }
48                                  bytes[bindex] = c[cindex+1];
49                                  cindex += 2;
50                                  bindex += 1;
51                          } else {
52                          throw new Exception("Unknown record type");
53                  }
54          }
55          return new String(bytes, 0, bindex, "UTF-8");
56  }
57
58  static boolean doit(byte[] c, int cindex, String dictionary, byte[] b, int bindex, int
        seqlen) {
59          if (cindex + seqlen <= c.length) {
60                  int dictindex = dictionary.indexOf(new String(c, cindex, seqlen));
61                  if (dictindex >= 0) {
62                          int constructed = 0x00008000;
63                          constructed |= dictindex << 3;
64                          constructed |= (seqlen-4) / 2;
65                          b[bindex] = (byte) ((constructed >>> 8) & 0xFF);
66                          b[bindex+1] = (byte) (constructed & 0xFF);
67                          return true;
68                  }
69          }
70          return false;
71  }
72
73  public static byte[] encode(String s, String dictionary) throws Exception {
74          byte[] b = new byte[2 * s.length()];
75          byte[] c = s.getBytes("UTF-8");
76          int cindex = 0;
77          int bindex = 0;
78          int freqs[] = new int[30];
```

```
79          for (int i = 0; i < 30; i++) { freqs[i] = 0; }
80          while (cindex < c.length) {
81                  boolean foundone = false;
82                  for (int j = 7; j >= 0; j--) {
83                          int seqlen = 2 * j + 4;
84                          if (doit(c, cindex, dictionary, b, bindex, seqlen)) {
85                                  cindex += seqlen;
86                                  bindex += 2;
87                                  freqs[seqlen]++;
88                                  foundone = true;
89                                  break;
90                          }
91                  }
92                  if (foundone) continue;
93                  if ((cindex + 2 <= c.length) && ((c[cindex] >= 0x00) && (c[cindex] < 0
                        x7F)) && ((c[cindex+1] >= 0x00) && (c[cindex+1] < 0x7F))){
94                          b[bindex] = (byte) (c[cindex] >> 1);
95                          b[bindex] &= 0x3F;
96                          b[bindex+1] = (byte) c[cindex+1];
97                          b[bindex+1] &= 0x7F;
98                          b[bindex+1] |= (byte) (c[cindex] << 7);
99                          bindex += 2;
100                         cindex += 2;
101                         freqs[2]++;
102                 } else {
103                         b[bindex] = (byte) 0x40;
104                         b[bindex+1] = (byte) c[cindex];
105                         bindex += 2;
106                         cindex += 1;
107                         freqs[1]++;
108                 }
109         }
110         byte[] r = new byte[bindex];
111         System.arraycopy(b, 0, r, 0, bindex);
112         System.out.printf("freqs: ");
113         for (int i = 0; i < 30; i++) {
114                 if (freqs[i] > 0) {
115                         System.out.printf("%dx%d, ", freqs[i], i);
116                 }
117         }
118         System.out.println();
119         return r;
120 }
121
122 public static void main(String args[]) {
123         if (args.length != 2) {
124                 System.err.println("Usage: Spaz dictionary input");
125                 return;
126         }
127         String dictionaryFilename = args[0];
128         String inputFilename = args[1];
129         try {
130                 String dictionary = new String(Files.readAllBytes(Paths.get(
                        dictionaryFilename)));
131                 String input = new String(Files.readAllBytes(Paths.get(inputFilename)));
132                 if (dictionary.length() > 4096) {
133                         throw new Exception("Dictionary is too long.");
134                 }
135                 byte[] b = Spaz2.encode(input, dictionary);
136                 String output = Spaz2.decode(b, dictionary);
137                 int inputLength = input.getBytes("UTF-8").length;
138                 int outputLength = b.length;
139                 double compression = ((double) inputLength) / ((double) outputLength);
140                 System.out.printf("Input length: %d; output length: %d; compression: %.3
                        fx\n", inputLength, outputLength, compression);
141                 if (!input.equals(output)) {
142                         throw new Exception("Strings do not match.");
143                 }
144                 } catch (Exception e) {
145                         e.printStackTrace();
146                 }
147         }
148 }
```

C Source codes of separating-secrets mitigation technique

C.1 Source code of separating-secrets experiment

```
1  import java.io.*;
2  import java.util.regex.Pattern;
3  import java.util.regex.Matcher;
4  import java.util.zip.GZIPOutputStream;
5  import java.util.zip.GZIPInputStream;
6  import java.nio.file.*;
7  public class exp{
8          /* method of separating secrets; separate contents assigned to the ``value"
               attribute*/
9          public static void SepSec(String original_file){
10                 try{
11                         FileReader filereader = new FileReader(original_file);
12                         BufferedReader in = new BufferedReader(filereader);
13                         FileWriter filewriter_ns = new FileWriter(original_file+"_ns");
14                         BufferedWriter bfr_ns = new BufferedWriter(filewriter_ns);
15                         FileWriter filewriter_s = new FileWriter(original_file+"_s");
16                         BufferedWriter bfr_s = new BufferedWriter(filewriter_s);
17                         Pattern pattern = Pattern.compile("value\\s*=\\s*\"([^\"]*)\"|
                               value\\s*=\\s*\'([^\']*)\'|value\\s*=\\s*[^\\s]*");
18                         Matcher p_matcher;
19                         String secret, nonsecret;
20                         String line = in.readLine();
21                         while (line != null){
22                                 nonsecret = line.replaceAll("value\\s*=\\s*\"([^\"]*)\"|
                                       value\\s*=\\s*\'([^\']*)\'|value\\s*=\\s*[^\\s]*", "
                                       FILTERED-OUT");
23                                 bfr_ns.write(nonsecret);
24                                 bfr_ns.newLine();
25                                 p_matcher = pattern.matcher(line);
26                                 while (p_matcher.find()) {
27                                         secret = p_matcher.group();
28                                                 bfr_s.write(secret);
29                                                 bfr_s.newLine();
30                                 }
31                                 line = in.readLine();
32                         }
33                         in.close();
34                         bfr_s.close();
35                         bfr_ns.close();
36                 }
37         catch(Exception e){
38                 System.out.print(e);
39         }
40 }
41
42 /*gzip compression; modified the code of Byron Kiourtzoglou (http://examples.
       javacodegeeks.com/core-java/io/fileinputstream/compress-a-file-in-gzip-format-in-
       java/)*/
43         public static void gzipFile(String source_filepath, String
               destinaton_zip_filepath) {
44                 byte[] buffer = new byte[1024];
45                 try {
46                         FileOutputStream fileOutputStream =new FileOutputStream(
                               destinaton_zip_filepath);
47                         GZIPOutputStream gzipOuputStream = new GZIPOutputStream(
                               fileOutputStream);
48                         FileInputStream fileInput = new FileInputStream(source_filepath)
                               ;
49                         int bytes_read;
50                         while ((bytes_read = fileInput.read(buffer)) > 0){
51                                 gzipOuputStream.write(buffer, 0, bytes_read);
52                         }
53                         fileInput.close();
54                         gzipOuputStream.finish();
55                         gzipOuputStream.close();
56                 }
57                 catch (Exception e){
58                         System.out.print(e);
```

```
59                  }
60          }
61
62  /*gunzip decompression; modified the code of Nikos Maravitsas (http://examples.
        javacodegeeks.com/core-java/io/fileinputstream/decompress-a-gzip-file-in-java-
        example/)*/
63          public static void unGunzipFile(String compressedFile, String decompressedFile){
64                  byte[] buffer = new byte[1024];
65                  try{
66                          FileInputStream fileIn = new FileInputStream(compressedFile);
67                  GZIPInputStream gZIPInputStream = new GZIPInputStream(fileIn);
68                          FileOutputStream fileOutputStream = new FileOutputStream(
                                decompressedFile);
69                  int bytes_read;
70                  while ((bytes_read = gZIPInputStream.read(buffer)) > 0) {
71                          fileOutputStream.write(buffer, 0, bytes_read);
72                  }
73                      gZIPInputStream.close();
74                  fileOutputStream.close();
75                  }
76                  catch (IOException ex){
77                  ex.printStackTrace();
78              }
79          }
80
81
82  /* method of recovering the original file from secrets and non-secrets*/
83          public static void MergeSec(String secret_file, String non_secret_file){
84                  try{
85                          FileWriter filewriter = new FileWriter(non_secret_file+"_recover
                                .htm");
86                          BufferedWriter out = new BufferedWriter(filewriter);
87                          FileReader filereader_ns = new FileReader(non_secret_file);
88                          BufferedReader bfr_ns = new BufferedReader(filereader_ns);
89                          FileReader filereader_s = new FileReader(secret_file);
90                          BufferedReader bfr_s = new BufferedReader(filereader_s);
91                          Pattern pattern = Pattern.compile("FILTERED-OUT");
92                          Matcher p_matcher;
93                          String line, secret;
94                          String recovery;
95                          StringBuffer sb;
96                          line = bfr_ns.readLine();
97                          while (line != null){
98                                  sb = new StringBuffer();
99                                  p_matcher = pattern.matcher(line);
100                                 while (p_matcher.find()) {
101                                         String text = p_matcher.group(0);
102                                         secret = bfr_s.readLine();
103                                         p_matcher.appendReplacement(sb, p_matcher.
                                                quoteReplacement(secret));
104                                 }
105                                 p_matcher.appendTail(sb);
106                                 recovery = sb.toString();
107                                 out.write(recovery);
108                                 out.newLine();
109                                 line = bfr_ns.readLine();
110                         }
111                         out.close();
112                         bfr_s.close();
113                         bfr_ns.close();
114                 }
115                 catch(Exception e){
116                         System.out.print(e);
117                 }
118         }
119
120         /*experiment*/
121         public static void main(String args[]){
122                 try{
123                         float original_size, compressed_size, mitigated_size,
                                secret_size, non_secret_size, avg1 = 0, avg2 = 0;
124                         FileReader filenames = new FileReader("names.txt");
125                         BufferedReader f_names = new BufferedReader(filenames);
```

```
126                         FileWriter w = new FileWriter("Report.text");
127                         BufferedWriter out = new BufferedWriter(w);
128                         String line = f_names.readLine();
129                         while (line != null){
130                                 String temp = new String(Files.readAllBytes(Paths.get(
                                        line)));
131                                 original_size = temp.getBytes("UTF-8").length;
132                                 gzipFile(line, line+".gzip");
133                                 compressed_size = Files.readAllBytes(Paths.get(line+".
                                        gzip")).length;
134                                 SepSec(line);
135                                 secret_size = Files.readAllBytes(Paths.get(line+"_s")).
                                        length;
136                                 gzipFile(line+"_ns", line+"_ns.gzip");
137                                 non_secret_size = Files.readAllBytes(Paths.get(line+"_ns
                                        .gzip")).length;
138                                 mitigated_size = non_secret_size + secret_size;
139                                 unGunzipFile(line+"_ns.gzip", line+"2_ns");
140                                 MergeSec(line+"_s", line+"2_ns");
141                                 out.write("File Name : "+line);
142                                 out.newLine();
143                                 out.write("-----------------------------");
144                                 out.newLine();
145                                 out.write("Size of the Original File : "+original_size+"
                                        bytes");
146                                 out.newLine();
147                                 out.write("Size of the Compressed File : "+
                                        compressed_size+" bytes");
148                                 out.newLine();
149                                 out.write("Compression : "+original_size/compressed_size)
                                        ;
150                                 avg1 += original_size/compressed_size;
151                                 out.newLine();
152                                 out.write("Extracter and Compressed : "+non_secret_size+"
                                        bytes");
153                                 out.newLine();
154                                 out.write("Size of the Secrets : "+secret_size+" bytes");
155                                 out.newLine();
156                                 out.write("Total size the mitigation : "+mitigated_size+"
                                        bytes");
157                                 out.newLine();
158                                 out.write("Compression : "+original_size/mitigated_size);
159                                 avg2 += original_size/mitigated_size;
160                                 out.newLine();
161                                 out.newLine();
162                                 line = f_names.readLine();
163                         }
164                         out.write("Avg Original Compression : "+avg1/10);
165                         out.newLine();
166                         out.write("Avg mitigated Compression : "+avg2/10);
167                         out.close();
168                 }
169                 catch (Exception e){
170                         System.out.print(e);
171                 }
172         }
173 }
```